Fine Art Search Engine

Rutuja Gurav, Krupa Hegde and Gautham Mani

Department of Computer Science, University of California, Riverside, CA, USA
{rgura001@ucr.edu, khegd001@ucr.edu, gmani001@ucr.edu}

1 Introduction

On the web there are several online stores to buy various kinds of Fine Art. Fine Arts aficionados visit a number of different websites before purchasing a painting. We sought to provide an one-stop interface to art lover for browsing paintings for sale across various online websites.

In this project, we built a search engine for Fine Art on sale on the web. We chose 5 online art stores selling art ranging from professional retailers to amateur artists. We crawled these websites to collect the data on paintings for sale and created an inverted index to rank and search paintings and present the user with top results from across these websites.

Section 2 gives the collaboration details of the members who have contributed to this project. Section 3 describes our methodology. Section 4 describes the Lucene index and search. Section 5 Technologies and Integration. Section 6 describes the Ranking strategy. Section 7 describes the User Interface. Section 8 discusses the limitations of this projects and the obstacles that we faced. Section 9 shows the runtime statistics of the Lucene and Hadoop index creation process. Section 10 is the conclusion.

2 Collaborations Details

1. Name: Gautam Mani
   Designed User interface for Fine arts search engine.
   Generated Json format index file from mapreduce index.
2. Name: Krupa Hegde
   Worked on query search and documents ranking model.
   Worked on Rest API.
3. Name: Rutuja Gurav
   Built inverted index in Hadoop using Map Reduce.

3 System Architecture

3.1 Architecture

The architecture of our system is shown in the figure 1 below. UI box indicates the web page of Fine Arts Search Engine. When the user queries, Restful API is

called based on the choices made either to call Lucene index or to call Hadoop index. Hadoop index and Lucene Index are stored in the database. Main class at the backend queries the database index collection. It fetched the relevant documents and computes ranks and gets the data from database collection where data is stored. These documents are sent back as the response. In further sections these processes are explained in detail.

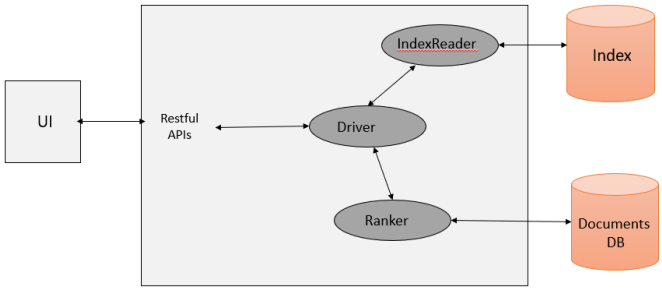

Fig. 1. Methodology

4 Overview of Hadoop Indexing and Ranking

We used Hadoop MapReduce to build an inverted index of our crawled data. The Hadoop Indexer's main task is to take as input our data in JSON format and produce an inverted index in JSON format. The indexer get essential keywords and generate index postings in a custom form that we have define in order to ensure better ranking.

The fields of the painting object and the indexing decisions that we took for them are mentioned in the table below.

Fields	Indexing Decision
URL	We did not use this field for indexing and hence it was not tokenized.
Description	Tokenized, stopwords and special characters removed and indexed.
Medium	Tokenized, stopwords and special characters removed and indexed.
Title	Tokenized, stopwords and special characters removed and indexed.
Color	JSON array of size 2, contains two dominant colors from the painting image. Tokenized, stopwords and special characters removed and indexed.
Tags	JSON array containing individual tags as values. Tokenized, stopwords and special characters removed and indexed.
Artist	Indexed.
Artist URL	Not used for indexing
Size	Not used for indexing
Price	Not used for indexing
Views	Not used for indexing
Likes	Not used for indexing
Image Source	Not used for indexing
Document ID	Not tokenized

We implemented 2 phase MapReduce to create an inverted index of our data.

The first phase mapper input was a LongWritable key and Text value. The value was a JSON Array containing JSON objects representing individual paintings. These JSON objects were parsed by the mapper using a JSON parser and appropriate field content was fetched. We then removed stopwords and special symbols from the field content and tokenized it into *words*. The output of the mapper was a key-value pair where the key was a combination of *word, field, Document ID* and value was 1. The first phase reducer then performed a simple aggregate wordcount on the mapper output and produced key-value pair output where the key is *word, field, Document ID* and the value is *count.*

The purpose of the second phase of MapReduce was to produce posting with *word* tokens as key and to bundle other details in the value. The second phase mapper took as its input the key-value pairs produced by the first phase reducer. The mapper function extracted the *word* which was the first token in the line. It produced key-value pairs where the key was the extracted *word* and the value was the remaining string in the line i.e. *field, Document ID, count.* The second phase reducer generated postings for each *word* and produced as its output key-value pairs where the key is *word* and the value is a list of tuples where each tuple contains *field, Document ID, count.*

While building index using Lucene, we did not have control over Lucene's internal BM25 ranking algorithm. By building a custom inverted index for our

data, we are able to rank the documents based on the importance of the field in which the query term occurs.

4.1 Ranking the Documents

After getting the relevant Document IDs from index. We considered few fields based on which we ranked the documents. As shown in the fig- our index in JSON format has the word field word. Its occurrences or frequency along with its location is stored. We have used a specific formula for our project to rank the documents instead of going for the existing ones. As we did, an image retrieval based on the text content of the image, the amount of text is less compared to the text retrieval. So, we modified the ranking formula as per our requirement.

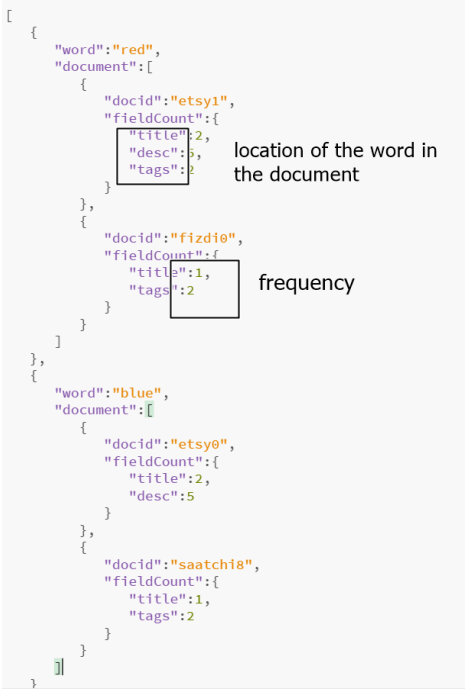

Fig. 2. Index

Our scoring method gives weights to individual fields. The index file contains the location i.e. the field where the *word* occurs and the frequency. We multiplied the frequency with weight assigned to the field. The figure below formalizes our shows our scoring formula.

The weights that we considered for each field are given in the table below.

$$\sum \log(Title * 10 + Tags * 8 + Artist * 7 + Medium * 5 + Desc * 4 + Color * 4)$$

Fig. 3. Scoring Formula

Fields	Weights
Title	10
Tags	8
Medium	5
Color	4
Description	4

We gave the field *medium* more weight than *description* because users usually tend to search for paintings using the *medium.* For example: "oil paintings of dog". After assigning weights, we multiplied it with the frequency from the retrieved documents and took a logarithm of the score to normalize the value. We then sorted the documents by their score in decreasing order.

5 Overview of Lucene indexing strategy

5.1 Fields in the Lucene index

Fields	Field Type	Justification
URL	StringField	We want it to be stored and indexed but not tokenized.
Description	TextField	Tokenized and indexed.
Medium	TextField	Tokenized and indexed.
Title	TextField	Tokenized and indexed.
Color	TextField	JSON array of size 2, contains two dominant colors from the painting image. Tokenized and indexed.
Tags	TextField	JSON array containing individual tags as values. Tokenized and indexed.
Artist	TextField	Tokenized and indexed.
Artist URL	StoredField	Stored, but not indexed or tokenized
Size	StoredField	Stored, but not indexed or tokenized
Price	StoredField	Stored, but not indexed or tokenized
Views	StoredField	Stored, but not indexed or tokenized
Likes	StoredField	Stored, but not indexed or tokenized
Image Source	StoredField	Stored, but not indexed or tokenized

Text analyzer choices The StandardAnalyser has a StandardTokenizer, StandardFilter, LowercaseFilter, and StopFilter.

We examined EnglishAnalyser which has some English stemming enhancements which work well for plain English text. However, since we are building an image search engine essentially based on tags and short descriptions, we do not have extensive text which would require such enhancements.

Standard Analyzer tokenizes cleverly the following lexical types: alphanumerics, acronyms, company names, e-mail addresses, computer host names, numbers, words with an interior apostrophe, serial numbers, IP addresses, and CJK (Chinese Japanese Korean) characters. StandardAnalyzer also includes stop-word removal, using the same mechanism as the StopAnalyzer which is sufficient for the purposes of our project.

We have not provided Analyzer options as an input argument while building the index because we are using only StandardAnalyzer by default.

5.2 Querying Lucene

Using IndexSearcher of Lucene we searched Lucene generated index files. We used Standard Analyzer and MultiQueryParser to query database on given query term. We considered the fields title, tags, description, and color for query parsing. We gave limit of top 10000 documents. We took the documentID from the query.

6 Technologies and Integration

6.1 User Interface

We developed the user interface using AngularJS and Twitter Bootstrap. AngularJS uses Model-View-Controller architecture. We wrote a common method in order to make a *GET* request to the API and called the API using AJAX. Certain parameters were used in order to distinguish between Lucene and Hadoop index. These parameters were sent to the back-end along with the API request to fetch the documents after an index lookup for the query word. The responses were in JSON format. We processed these with Angular template and rendered them to front end.

6.2 Back-end

We stored our index and data in MongoDB NoSQL database in 2 separate collections. We used the Spring framework to write RESTFUL services. We have 2 APIs, one for Lucene search and one for Hadoop search. By calling the Lucene search API, it takes the query word and looks it up in the Lucene index. By using Lucene QueryParser we retrieve the matched Document IDs. Similarly with Hadoop search API, it takes the keyword and queries the MongoDB *Index* collection. It retrieves matched documents from *Documents* collection and then computes the scores for those documents and sorts them in descending order or the scores. As Spring-Rest uses Jackson jsons, it automatically converts the list of documents to json format and sends it to the front-end. We also use ngrock to get a global address which can be called from anywhere to get the response.

7 Results

We built an image search engine for fine arts which is able to search over a collection of paintings and produce ranked results.

There are 2 options on our web-page to select either Lucene or Hadoop index to perform the search. See figure 5.

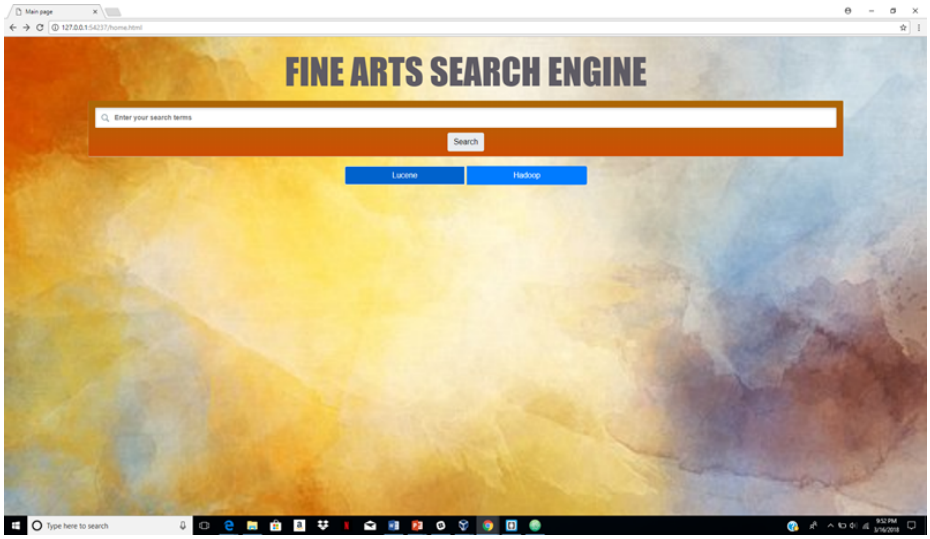

Fig. 4. home page of Fine arts search engine

7.1 Snippetting

We display a snippet which shows the top painting and suggests the user to visit the artist's website and to search for similar queries based on the tags of the top result. See figure 6.

Below the snippet, we display the results. Each result has painting title, artist's name, price and some cases a description. On clicking either title or image, it directs to the original page where the painting can be purchased. By clicking on artist's name, it takes the user to the artist's page. See figures 7 and 8.

The suggested search queries are based on the tags of the current top result. As can be seen from figure 9, for the query apple, fine art is one of the suggested query words. On clicking the suggested query it fetches results for that query and display it. See figure 9.

Our Hadoop index-based search is comparable with Lucene index-based search in terms of query results. Fig - shows the time taken for building Hadoop and

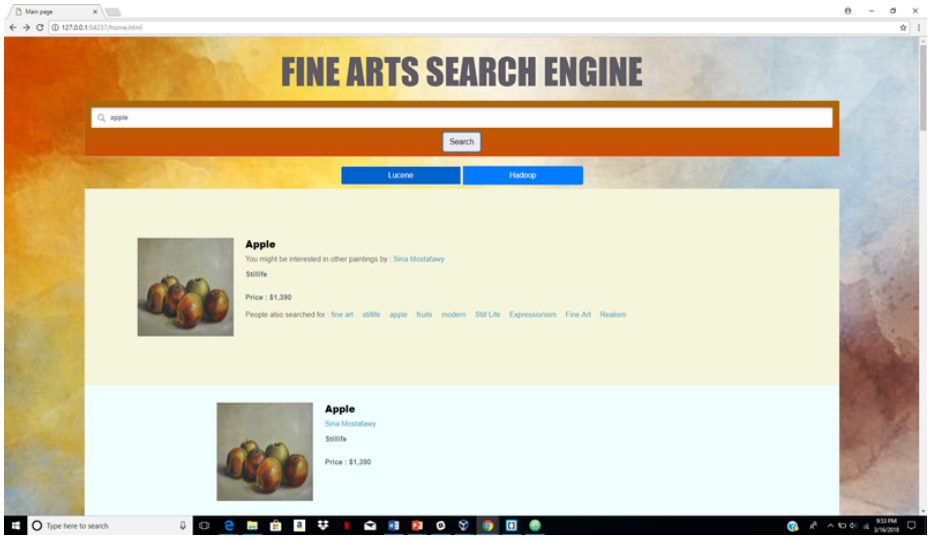

Fig. 5. Query Results with Snippets

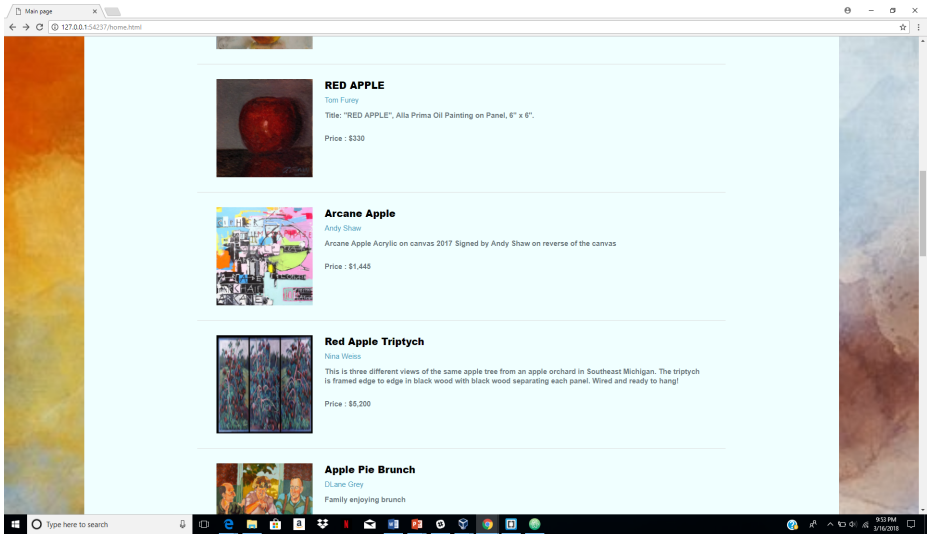

Fig. 6. Query Results for keyword-"apple"

Lucene index. We can clearly see on larger data Hadoop performs better than Lucene.

7.2 Runtime of Lucene and Hadoop index creation process

The runtime statistics of index creation are shown in the figure below - The

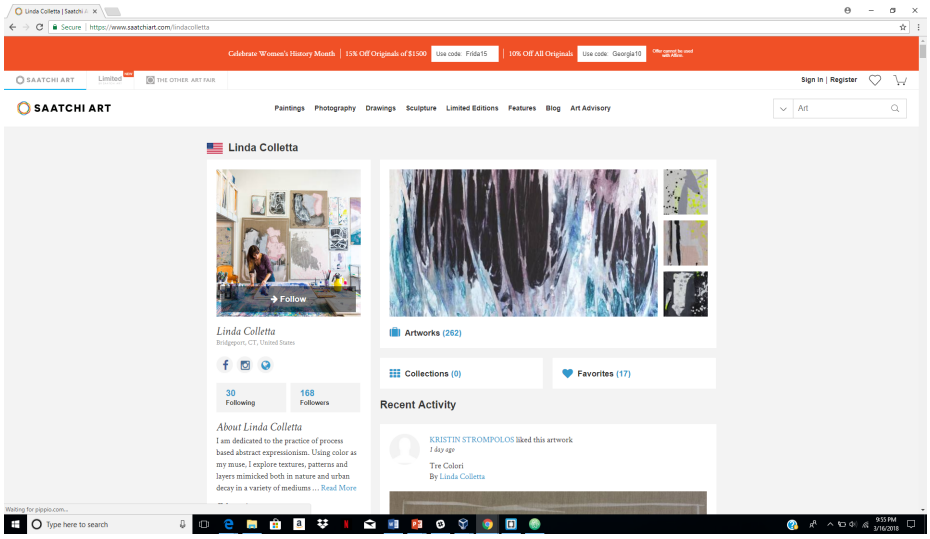

Fig. 7. Directing to artist's page

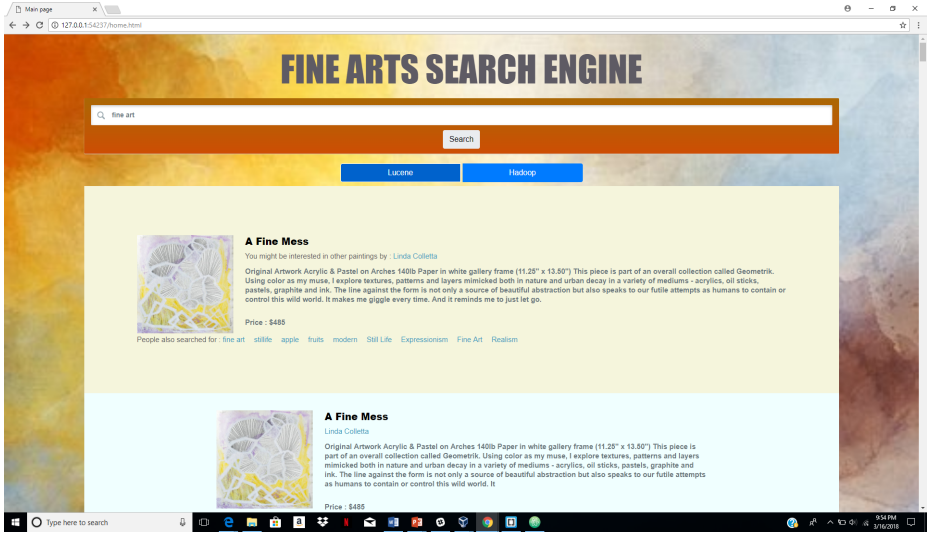

Fig. 8. Fetching results from suggested queries

Hadoop indexer takes comparably less time than Lucene indexer.

8 Limitations and Obstacles

Due to the nature of the data that we chose to crawl, we were unable to fully parallelize the crawling process as we were scrapping 5 different websites with

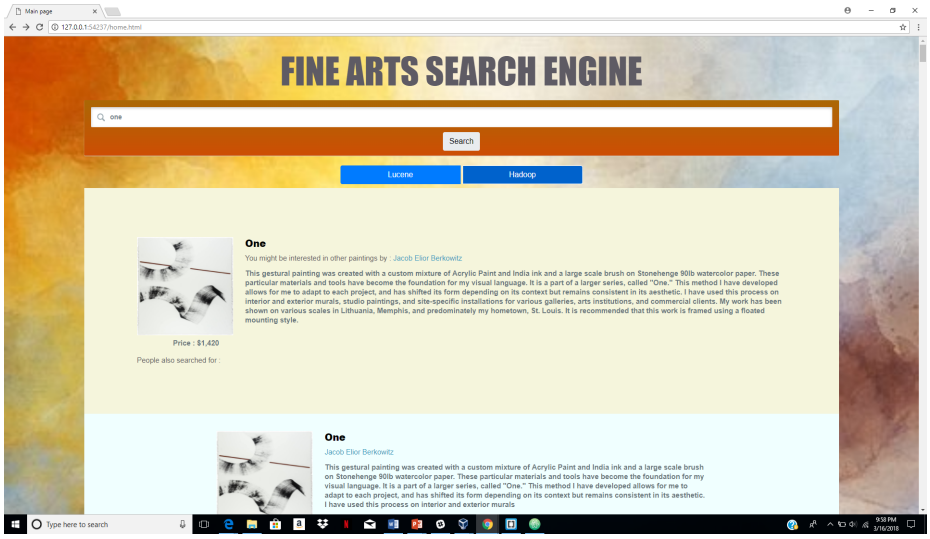

Fig. 9. Query Results from hadoop search

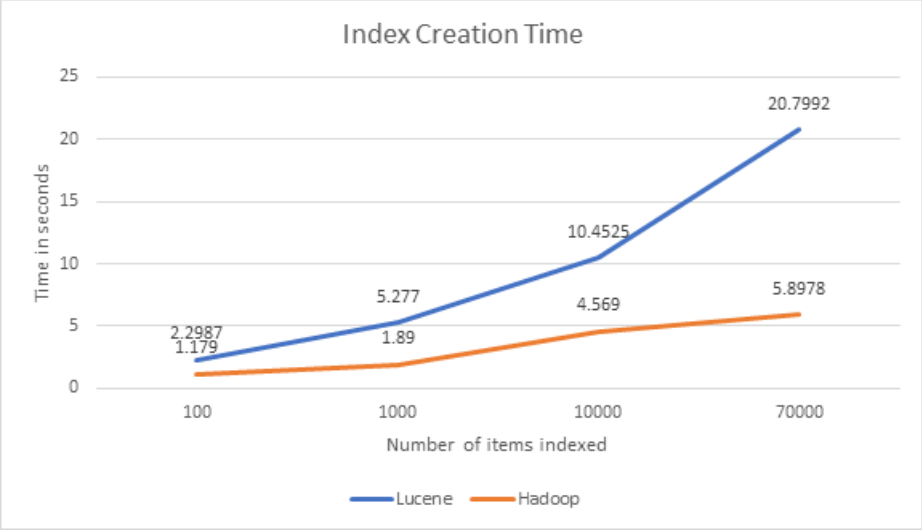

Fig. 10. Runtime Statistics

different DOM structures. Since the request were to the same website again and again and we had to maintain delays we did not create new threads for each new URL. Instead, we multi-threaded the crawling process for the 5 websites by dedicating one thread per website.

References

1. Cookbook:JSoup Java HTML parser: https://jsoup.org/cookbook/
2. Apache Lucene Guide: https://lucene.apache.org/
3. MatLab Guide for Image Processing: https://www.mathworks.com/help/images/getting-started-with-image-processing-toolbox.html
4. Book by Erik Hatcher and Otis Gospodneti (2004): Lucene in Action
5. General Syntaxes: https://stackoverflow.com/
6. Json-simple: https://code.google.com/archive/p/json-simple/
7. Spring-rest: http://spring.io/guides/gs/accessing-mongodb-data-rest/
8. AngularJS: https://www.w3schools.com/angular/default.asp/
9. Hadoop-MapReduce: https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html